Natural Language Processing

Lab 6: Edit Distance and Spell Correction

Objectives

1. To implement the dynamic programming algorithm for calculating the Levenshtein (Minimum Edit) Distance between two strings.
2. To understand how this distance can be used to model the likelihood of a misspelling.
3. To build a simple, non-probabilistic spell corrector that suggests corrections for a misspelled word from a given vocabulary based on minimal edit distance.

The Problem of Spelling Errors

Spelling errors are ubiquitous in human-generated text. A spell correction system must perform two key tasks:

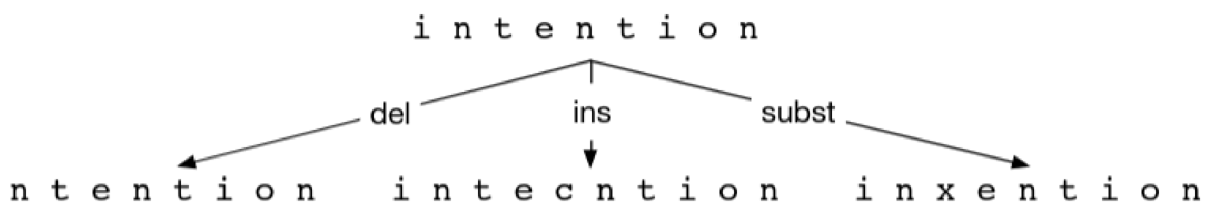
1. **Error Detection:** Identify which words in a text are misspelled.
2. **Error Correction:** Suggest the most probable correct word(s).

A simple way to detect an error is to check if a word exists in a pre-defined dictionary (vocabulary). If it does not, it is flagged as a potential error. This lab focuses on the correction task.

Edit Distance

The core idea is that the correct word is often the one that is most "similar" to the misspelled word. We measure this similarity using **Edit Distance**.

Definition: The **Minimum Edit Distance** between two strings is the minimum number of **primitive operations** required to convert one string into the other.



The standard set of primitive operations (with an associated cost) is:

- **Insertion** (cost = 1): Insert a character. (e.g., "cat" -> "cart")
- **Deletion** (cost = 1): Delete a character. (e.g., "cat" -> "at")

- **Substitution** (cost = 1 or 2): Replace one character with another. (e.g., "cat" -> "bat"). A cost of 2 is often used to make substitutions less favorable than an insertion+deletion pair.

When all operation costs are assigned to edit distances, this is known as the **Levenshtein Distance**.

```
I N T E * N T I O N
| | | | | | | | | |
* E X E C U T I O N
d s s   i s
```

Minimum Edit Distance Algorithm

$$D[i,j] = \min \begin{cases} D[i-1,j]+1 \\ D[i,j-1]+1 \\ D[i-1,j-1] + \begin{cases} 2; & \text{if } source[i] \neq target[j] \\ 0; & \text{if } source[i] = target[j] \end{cases} \end{cases}$$

```
**function** MIN-EDIT-DISTANCE(*source*, *target*) **returns** *min-distance*

    $n \leftarrow$ LENGTH(*source*)
    $m \leftarrow$ LENGTH(*target*)
    Create a distance matrix $D[n+1,m+1]$

    # Initialization: the zeroth row and column is the distance from the empty string
    $D[0,0] = 0$
    **for** each row $i$ **from** 1 **to** $n$ **do**
        $D[i,0] \leftarrow D[i-1,0] + \textit{del-cost}(\textit{source}[i])$
    **for** each column $j$ **from** 1 **to** $m$ **do**
        $D[0,j] \leftarrow D[0,j-1] + \textit{ins-cost}(\textit{target}[j])$

    # Recurrence relation:
    **for** each row $i$ **from** 1 **to** $n$ **do**
        **for** each column $j$ **from** 1 **to** $m$ **do**
            $D[i,j] \leftarrow$ MIN( $D[i-1,j] + \textit{del-cost}(\textit{source}[i])$,
                                 $D[i-1,j-1] + \textit{sub-cost}(\textit{source}[i], \textit{target}[j])$,
                                 $D[i,j-1] + \textit{ins-cost}(\textit{target}[j])$)
    # Termination
    **return** $D$[n,m]
```

Src\Tar	**#**	**e**	**x**	**e**	**c**	**u**	**t**	**i**	**o**	**n**
#	0	1	2	3	4	5	6	7	8	9
i	1	2	3	4	5	6	7	6	7	8
n	2	3	4	5	6	7	8	7	8	7
t	3	4	5	6	7	8	7	8	9	8
e	4	3	4	5	6	7	8	9	10	9
n	5	4	5	6	7	8	9	10	11	10
t	6	5	6	7	8	9	8	9	10	11
i	7	6	7	8	9	10	9	8	9	10
o	8	7	8	9	10	11	10	9	8	9
n	9	8	9	10	11	12	11	10	9	8

Task 1

Implement the **MIN-EDIT-DISTANCE** function from the pseudocode. Use the Levenshtein Distance

Task 2

Create a command line application that prints out top 10 correct words for an invalid word in a string without using any external libraries.

Task 3

Create a command line application that prints out top 10 suggested words as you type without using any external libraries.